Journal of Algorithms **42,** 205–230 (2002)
doi:10.1006/jagm.2002.1211, available online at http://www.idealibrary.com on IDEAL®

Randomized Sorting in $O(n \log\log n)$ Time and Linear Space Using Addition, Shift, and Bit-wise Boolean Operations [1, 2]

Mikkel Thorup

AT&T Labs—Research, Shannon Laboratory, 180 Park Avenue, Florham Park, New Jersey 07932
E-mail: mthorup@research.att.com

Received June 11, 1997

A randomized sorting algorithm is presented, doing as described in the title. Implications of the techniques are discussed for dictionaries and priority queues.

Key Words: sorting; priority queues; batched dictionaries; word RAM.

1. INTRODUCTION

In this paper we consider sorting on a very simple RAM where the only word operations are addition, shift, and bit-wise boolean operations. Besides these word operations, we have direct and indirect addressing, jumps, and conditional statements. Such a RAM has been referred to as a *Practical RAM* [14]. In this paper we show

THEOREM 1. *On a Practical RAM, there is a randomized algorithm sorting n words in $O(n \log\log n)$ time and linear space.*

Here time refers to the number of operations performed; that is, all operations are considered of constant cost. Note that the bound is independent

[1] A preliminary version of this paper was presented at SODA'97 [21]. The preliminary version was done while the author was at University of Copenhagen.

[2] Authors of selected papers from SODA'97, judged to be among the best by the program committee in their initial ratings, were invited to submit journal versions of their papers to a special issue of *Journal of Algorithms*. These submissions underwent the standard review process for this journal. This paper is one of the selected papers and should have appeared with the others in Volume 30, Number 2, February 1999.

0196-6774/02 $35.00

of word length w. All we assume about w is that it is larger than $\log_2 n$; otherwise, we would not be able to refer uniquely to the different keys to be sorted. If $w \gg \log n$, it means that the domain $\{\underline{0}, \underline{1}\}^w$ of words is very large. At the same time, it means that word operations have the power of operating on comparatively many bits in parallel. The algorithm of Theorem 1 only makes shifts by powers of two, and it only needs $O(\log n)$ random words.

Our time bound matches that of the current fastest sorting algorithm by Andersson *et al.* [2]. Their algorithm has two variants: one is deterministic and uses space $2^{\varepsilon w}$, where w is the word length and ε is a positive constant. Thus the space is unbounded in terms of n. The other variant is randomized and uses linear space like ours, but it uses multiplication.

Multiplication is considered expensive from several perspectives. From a theoretical viewpoint, it is known that the minimal depth of a polynomial-size circuit implementing multiplication is $\Theta(\log w/\log\log w)$ [5], which is unbounded in n. In contrast, all the Practical RAM operations are in AC^0; that is, they are all implementable by polynomial-size constant depth circuits. From a practical viewpoint, we may circumvent the theoretical lower-bound by pipelining several multiplications at the time. Speed may be gained by tabulating the multiplication of small numbers and by having many adders working in parallel. Nevertheless, multiplication takes a longer time on most computers than the AC^0 operations of the Practical RAM.

A more serious objection to multiplication is that the cost of implementing fast multiplication grows rapidly with the word size; e.g., for addition based multiplication, we could use $\Theta(w)$ adders working in parallel. Thus, insisting on fast multiplication could be prohibitive for increasing the word length. Conversely, restricting ourselves to the Practical RAM operations would make it easier to increase the word length, hence the potential for exploiting processor bit-parallelism. Theorem 1 states that with our current knowledge, the restriction to Practical RAMs does not have any asymptotic influence on the running time of sorting. Hence Practical RAMs could be used for future sorting coprocessors with large word length.

A trend toward increasing the word length without full-length multiplication is already happening in modern processors such as Intel's Pentium 4 [12] (see [20] for a survey of such processors). Besides handling standard 64-bit registers, the Pentium 4 offers a limited instruction set for dealing with 128-bit registers. This instruction set is not quite that of the Practical RAM, but nevertheless, it is very suitable for implementing the algorithms presented in this paper.

The problem of doing $o(n \log n)$ sorting with AC^0 operations was posed by Fredman and Willard [10] when they presented the first $o(n \log n)$ sorting algorithms. Both their deterministic $O(n \log n/\log\log n)$ algorithm and

their randomized $O(n\sqrt{\log n})$ algorithm were heavily based on multiplication. The above mentioned deterministic $O(n \log\log n)$ sorting algorithm of Andersson *et al.* [2] avoids multiplication, but uses space unbounded in terms of n. Since then, Raman [18] has found a deterministic Practical RAM algorithm running in time $O(n\sqrt{\log n \log\log n})$ and linear space. Also, Andersson *et al.* [3] have presented a randomized linear space sorting algorithm with running time $O(n(\log\log n)^2)$. However, that algorithm is based on some powerful nonstandard AC^0 operations. For example, for the "cluster busting" in [3, Lemma 2], they assume an operation that takes a word x and w other words, $a_0, ..., a_{w-1}$ and computes the bit-wise 'or' of all a_i for which bit i is set in x. As stated in Theorem 1, in this paper we reduce the running time to $O(n \log\log n)$ and use standard AC^0 operations only.

Techniques

The previous randomized linear space $O(n \log\log n)$ sorting algorithm of Andersson *et al.* [2] makes extensive use of multiplication based hashing. More precisely, it uses hashing to identify duplicates in a list of keys, but we note here that any duplicate finding technique would do:

LEMMA 1 [2]. *On a Practical RAM, we can sort in time* $O(n + \sum_{i=0}^{\log\log n} \times D(n, w/2^i))$ *where* $D(n, b)$ *is the time it takes to identify duplicates in a list of* n b*-bit keys. The space consumption is the same as that for identifying the duplicates.*

If we use universal hashing [7] as in [2], the identification of duplicates is done in linear time on the unit cost RAM: simply scan the keys one by one, adding each key to the list of previous identical keys, if any. However, it follows from [3] that if we restrict ourselves to AC^0 operations, then any such "on-line" approach for finding duplicates, where the on-line algorithm must process the words one-at-a-time, and cannot undo previously made decisions, is going to take time $\Omega(n\sqrt{\log n/\log\log n})$. In fact, the lower-bound holds more generally for any instruction set if we say that the cost of an operation is the minimal depth of a polynomial-size circuit computing it. On the other hand, in [3] it is shown that the real "off-line" duplicates problem, where one is allowed to look at all of the words before doing anything, can be solved in time $O(n \log\log n)$ using some powerful nonstandard AC^0 operations. Using Lemma 1, this leads to an $O(n(\log\log n)^2)$ AC^0 sorting algorithm. In this paper, we show

THEOREM 2. *We can identify the duplicates in a list of* n (w/q)*-bit keys in* $O(n + n \log\log n/q)$ *expected time by a randomized Practical RAM algorithm using linear space.*

Together with Lemma 1, this immediately implies Theorem 1.

Other Results

Our sorting techniques have implications for priority queues and dictionaries. A basic *priority queue* is a dynamic set Q of integer keys for which we can insert a key x with `insert`(x, Q), find the minimum key with `find-min`(Q), and delete the minimum key with `delete-min`(Q). The priority queue is said to be *monotone* if the minimum is nondecreasing— the minimum of an empty monotone priority queue is defined to be $-\infty$. This restriction does not affect most greedy algorithms.

COROLLARY 1. *There is a randomized monotone Practical RAM priority queue supporting n* `find-min` *in constant time, and* `insert` *and* `delete-min` *in $O(\log\log n)$ expected amortized time. Here n is the current number of keys in the queue.*

Proof. From [22, Theorem 1.4] we have a general reduction from amortized monotone priority queues to sorting. The version for the Practical RAM states that if we can sort n keys in time $n \cdot s(n)$ where s is a nondecreasing function, then we can support n operations on a monotone queue, starting empty, in time $O(n \sum_{i=0}^{\log^* n} s(\log^{(i)} n))$. Here $\log^{(0)} n = n$ and $\log^{(j+1)} = \log(\log^{(j)} n)$, and $\log^* n$ is the least j such that $\log^{(j)} n \leq 1$. From Theorem 1, we have $s(n) = \log\log n$, and then $\sum_{i=0}^{\log^* n} s(\log^{(i)} n) = O(\log\log n)$. ∎

Plugging the above monotone priority queue into Dijkstra's single source shortest path algorithm [9], we get

COROLLARY 2. *On the Practical RAM, we can solve the single source shortest path problem on a graph with m integer weighted edges in $O(m \log\log m)$ expected time.*

Our techniques developed for fast linear space Practical RAM sorting also have consequences for dictionaries. A *dictionary* is a dynamic set for which we can insert elements, delete elements, and ask for membership. Using multiplicative hashing [7], we can support each of the above operations in constant expected time per operation. However, on the Practical RAM, where multiplication is not available, the problem is provably much harder. In [3] it is shown that there are concrete static sets X of words, such that even if we allow arbitrarily AC^0 operations and polynomial space, verifying membership of X takes $\Omega(\sqrt{\log n / \log\log n})$ time on the average. In particular this lower bound holds for the Practical RAM. However, in this paper, we show

THEOREM 3. *On a Practical RAM, we can process a batch of $\log n$ operations on a dynamic dictionary with n keys in $O(n)$ space and $O((\log n)(\log\log n)^{1+\varepsilon})$ expected time.*

Thus, batching of just a few operations gives us an exponential improvement in the per operation cost of dictionaries on the Practical RAM. The result for batched dictionaries is then used to make fast general priority queues where the minimum may decrease.

THEOREM 4. *There is a randomized Practical RAM general priority queue that with n elements supports* `insert`, `delete`, *and* `extract-min` *in* $O((\log \times \log n)^{1+\varepsilon})$ *expected time*.

In view of the above mentioned negative results from [3], informally we believe that priority queues capture just about the most general operations that can be supported on a dynamic set in doubly logarithmic time per operation on the Practical RAM.

Contents

After a preliminary section, Theorem 2 is proved in Sections 3–5. Theorem 3 is proved in Section 6 and Theorem 4 is proved in Section 7. In Section 8 we briefly discuss implementations with a Pentium 4. Finally, we conclude with some open problems in Section 9.

2. PRELIMINARIES

In this paper, log is understood to be base 2. Also, most numbers are implicitly assumed to be powers of two so that we avoid rounding problems.

The paper is primarily concerned with manipulation of bit-strings. We use multiplication notation for concatenation of strings. In order to avoid ambiguity, we use $\underline{0}$ and $\underline{1}$ for bits that are part of bit-strings. Thus $\underline{1} \cdot \underline{0} = \underline{10}$ while $1 \cdot 0 = 0$. Consider a bit string $x = b_0 \cdots b_{l-1}$, $b_i \in \{\underline{0}, \underline{1}\}$. Then $|x| = l$ and $x[i]$ denotes b_i if $i < |x|$; $\underline{0}$ otherwise. Also $x[i..j]$ denotes $x[i] \cdots x[j]$. Moreover, the number of $\underline{1}$s in a string x is denoted $\|x\|_1$. If $\|x\|_1 = 0$, we say that x is *empty*. For bit-strings x and y of equal length, we call $\|x \oplus y\|_1$ the *Hamming distance* from x to y. The set $\{y \mid \|x \oplus y\|_1 \leq r\}$ is called the *Hamming ball* with radius r and center x.

We will typically view our bit-strings as vectors of k-bit fields, and then $x\langle i \rangle_k$ denotes field i; that is,

$$x\langle i \rangle_k = x[ik..(i+1)k - 1].$$

For example, if x is divided over multiple words, the first word is found as $x\langle 0 \rangle_w$.

The operators $\wedge$, $\vee$, $\neg$, and $\oplus$ denote bit-wise 'and', 'or', 'not', and 'exclusive-or', respectively. Also, we have $x \ll d$ and $x \gg d$ denoting standard left and right shift, respectively. That is, $(x \ll d)[i] = x[i + d]$ if

$i < |x| - d$; $\underline{0}$ otherwise. We note that all these operations are defined for bit-strings of arbitrary length, and their time complexity is $O(\lceil |x|/w \rceil)$.

In order to avoid ambiguity, we will use explicit casting between integers and strings. More precisely, $\texttt{integer}(x)$ denotes the number represented by the string x; i.e. $\texttt{integer}(x) = \sum_{i=1}^{|x|} 2^{|x|-i}\texttt{integer}(x[i])$, where $\texttt{integer}(\underline{0}) = 0$ and $\texttt{integer}(\underline{1}) = 1$. Note that our bit-strings are viewed as lexicographically ordered, with the most significant bit coming first. Similarly, $\texttt{string}(i, l)$ denotes the string x of length l representing i in the sense that $\texttt{integer}(x) = i \bmod 2^l$. Note that if l is not long enough to represent i, we discard the most significant bits.

If the arithmetic operations $+$ and $\times$ are applied to equal length bit strings x and y, they are implicitly converted to integers and back, producing a bit-string of the same length, that is, $x + y$ denotes $\texttt{string}(\texttt{integer}(x) + \texttt{integer}(y), |x|)$ and $x \times y$ denotes $\texttt{string}(\texttt{integer}(x) \times \texttt{integer}(y), |x|)$. So far, we have not assumed minus in our instruction set, but it can always be computed as $-x = \neg x + \texttt{string}(1, |x|)$.

Note that if x and y are bit-strings, then xy and $x \cdot y$ denote their concatenation whereas $x \times y$ denotes their multiplication.

As an exercise in our notation, we implement a standard routine that we will use frequently. A similar implementation routine can for example be found in [1].

ALGORITHM A. $\texttt{Mask}(x, k)$. Returns y where $y\langle i \rangle_k$ equals $\underline{1}^k$ if $x\langle i \rangle_k \neq \underline{0}^k$; $\underline{0}^k$ otherwise.

A.1 $x_1 := (x \wedge (\underline{0}\,\underline{1}^{k-1})^{|x|/k}) + (\underline{0}\,\underline{1}^{k-1})^{|x|/k}$
—$x_1\langle i \rangle_k[0] = \underline{1} \iff x_1\langle i \rangle_k[1..k-1] \neq \underline{0}^{k-1}$

A.2 $x_2 := (x \vee x_1) \wedge (\underline{1}\,\underline{0}^{k-1})^{|x|/k}$
—$x_2\langle i \rangle_k$ equals $\underline{1}\,\underline{0}^{k-1}$ if $x\langle i \rangle_k \neq \underline{0}^k$; $\underline{0}^k$ otherwise

A.3 $x_3 := x_2 \gg k - 1$
—$x_3\langle i \rangle_k$ equals $\underline{0}^{k-1}\,\underline{1}$ if $x\langle i \rangle_k \neq \underline{0}^k$; $\underline{0}^k$ otherwise

A.4 $x_4 := x_3 + (\underline{0}\,\underline{1}^{k-1})^{|x|/k}$
—$x_4\langle i \rangle_k$ equals $\underline{1}\,\underline{0}^{k-1}$ if $x\langle i \rangle_k \neq \underline{0}^k$; $\underline{0}\,\underline{1}^{k-1}$ otherwise

A.5 Return $x_4 \oplus (\underline{0}\,\underline{1}^{k-1})^{|x|/k}$

Assuming that the relevant prefixes of $(\underline{0}\,\underline{1}^{k-1})^\infty$ and $(\underline{1}\,\underline{0}^{k-1})^\infty$ are precomputed, the algorithm only makes a constant number of string operations. If x fits in one word, $\texttt{Mask}(x, k)$ is computed in constant time. Otherwise, suppose that $|x| > w$ but $k \leq w$ where w is the word length. Assuming that x is distributed over $q = |x|/w$ words $x_1, \ldots, x_q$, we simply compute $\texttt{Mask}(x, k)$ as $\texttt{Mask}(x_1, k) \cdots \texttt{Mask}(x_q, k)$. Here, we are exploiting that the carry in step A.4 does not propagate between pieces. Conversely, suppose that we are considering several keys $x_1, \ldots, x_q$, $|x_i| = w/q$,

$q = \omega(1)$, packed in one word; then $\texttt{Mask}(x_1, k) \cdots \texttt{Mask}(x_q, k)$ may be computed as $\texttt{Mask}(x_1 \cdots x_q, k)$. Thus, provided that keys are packed in words, the amortized cost of computing $\texttt{Mask}(x_i, k)$ is $O(|x_i|/w) = o(1)$.

We will now argue that our sorting algorithm is able to produce relevant prefixes of $(\underline{1}\,\underline{0}^{k-1})^\infty$ and $(\underline{0}\,\underline{1}^{k-1})^\infty$ in a preprocessing step. For our time bounds to hold, it suffices to know the first word of each of them, i.e., $(\underline{1}\,\underline{0}^{k-1})^{w/k}$ and $(\underline{0}\,\underline{1}^{k-1})^{w/k}$, and since one is the negation of the other, it suffices to know $(\underline{1}\,\underline{0}^{k-1})^{w/k}$.

ALGORITHM B. Constructs $(\underline{1}\,\underline{0}^{k-1})^{w/k}$ by doubling.

B.1. $x := \texttt{string}(1, w) \ll w - 1$

B.2. $b := k$

B.3. while $b < w$, —$x = (\underline{1}\,\underline{0}^{k-1})^{b/k}\underline{0}^{w-b}$

B.3.1. $x := x \wedge (x \gg b)$

B.3.2. $b := 2b$

Clearly, the cost of the above computation is $O(\log q)$ where $q = w/k$. In our sorting application, w is unbounded in the number of elements n, but we will always have $q = (\log n)^{O(1)}$, so the above computation takes time $O(\log \log n)$. Also, w and k, and hence q, will be powers of 2, so $q = (\log n)^{O(1)}$ implies that q can only attain $O(\log \log n)$ different values. Hence the computation of all the relevant "constants" is done as a preprocessing in total time $O((\log \log n)^2) = o(n)$. The above type of reasoning is considered straightforward, and it will often be assumed implicitly in the rest of this paper.

3. THE GENERAL APPROACH

We will borrow the general approach from [3]. We are starting with a sequence X of n keys, each taking up one word. Our aim is to shorten the keys, so that they can be sorted in linear time (sorting implies that we group duplicates); either by packed sorting [1, 2] or by radix sort with character of length $\log n$, depending on the relationship between n and w.

First, we have a *clustering phase* where we compute a short *signature* of each key so that with high probability, if two keys get the same signature, they have small Hamming distance. Using sorting on the short signatures, we cluster keys with the same signatures into Hamming balls of small radius.

Next, we have a *cluster busting phase*. For each cluster Y of keys with the same signature, we select an arbitrary key $y_0 \in Y$ and compute the set $Y \oplus y_0 = \{y \oplus y_0 | y \in Y\}$. If r is the maximal difference between any two keys in Y, then no key in $Y \oplus y_0$ contains more than r $\underline{1}$s; i.e., $\forall\, y \in Y : \|y \oplus y_0\|_1 \leq r$.

We then need to develop methods reducing keys with few $\underline{1}$s injectively to keys that are short enough for linear time sorting.

All the sets $Y \oplus y_0$ may be busted simultaneously. The shortest keys we will consider are of length $w/(\log n)^{\Theta(1)}$, so, essentially, we can always assume that we have enough keys to fill the words.

4. LONG WORDS

In this section, we prove Theorem 2 under the following assumption:

$$\log w = \omega(\log\log n) \iff w = (\log n)^{\omega(1)}. \tag{1}$$

As pointed out in [2], the results from [1] imply

LEMMA 2 [1]. *On an Practical RAM we can sort n keys of length $O(w/(\log n \log\log n))$ in time $O(n)$ and space $O(n)$.*

Hence, in this section, the goal is to reduce the length of our keys to $O(w/(\log n \log\log n))$. Thus, assume that our keys have length $b = \omega(w/(\log n \log\log n))$. Set $k = w/(\log n \log\log n)$, and $q = b/k$. Let α be a random string of b bits (each bit is independently set to $\underline{1}$ with probability 1/2). A signature of length k is computed as follows.

ALGORITHM C. $\mathtt{Sig}_\alpha(x)$.

C.1. $y := x \wedge \alpha$.

C.2. Return $\bigoplus_{i=1}^{q} y\langle i\rangle_k$.

The following lemma states that our signature separates keys that differ in many bit-positions.

LEMMA 3. *For any two words x and y, if α is randomly chosen and $\|x \oplus y\|_1 = r$, then* $\Pr(\mathtt{Sig}_\alpha(x) = \mathtt{Sig}_\alpha(y)) \le 2^{-r/q}$.

Proof. Let $z = x \oplus y$. Then $\mathtt{Sig}_\alpha(z) = \mathtt{Sig}_\alpha(x) \oplus \mathtt{Sig}_\alpha(y)$. We are searching the probability that $\mathtt{Sig}_\alpha(z) = \underline{0}^k$.

For $j = 1, \ldots, k$, set $I(j) = \{i | z\langle i\rangle_k[j] = \underline{1}\}$. Then $\mathtt{Sig}_\alpha(z)[j] = \bigoplus_{i \in I(j)} \alpha\langle i\rangle_k[j]$. Since each $\alpha\langle i\rangle_k[j]$ is independently chosen to be $\underline{1}$ with probability 1/2, it follows that $\Pr(\mathtt{Sig}_\alpha(z)[j] = \underline{1}) = 1/2$ if $I(j) \neq \emptyset$; 0 otherwise. Since $\|z\|_1 = r$ and $|I(j)| \le q$ for all j, it follows that $I(j) \neq \emptyset$ for at least r/q values of j. ∎

We will now show that, in fact, we can compute all the signatures in linear time. The basic idea is this. We will compute $\bigoplus_{i=1}^{q} y\langle i\rangle_k$ by repeatedly $\oplus$ing the two halves of y. This halves the size of y, and we can benefit from that if we work on the list of all keys simultaneously. Below follows the formal code, which we then explain afterward.

ALGORITHM D. `ParallelXOR`$(\vec{y}, b, k)$ where $\vec{y} = y_0 \cdots y_{n-1}$ is a packed list of b-bit keys, k and b powers of two, and $k < b$. The algorithm computes the list

$$\left(\left(\bigoplus_{i=1}^{b/k} y_0\langle i\rangle_k\right)\underline{0}^{b-k}\right)\cdots\left(\left(\bigoplus_{i=1}^{b/k} y_{n-1}\langle i\rangle_k\right)\underline{0}^{b-k}\right)$$

in $O(|\vec{y}|/w + \log(b/k))$ time.

D.1. $b' := b$

D.2. While $b > k$:

D.2.1. $b := b/2$

D.2.2. $\vec{y} := (\vec{y} \wedge (\underline{1}^b\underline{0}^b)^{|\vec{y}|/(2b)}) \oplus ((\vec{y} \wedge (\underline{0}^b\underline{1}^b)^{|\vec{y}|/(2b)}) \ll b)$;

D.2.3. `Collect`$(\vec{y}, b)$

D.3. While $b < b'$: `Spread`$(\vec{y}, b)$; $b := 2b$.

D.4. Return $\vec{y}$.

Here

$$\texttt{Spread}(x, k) \equiv (x \wedge (\underline{1}^k\underline{0}^k)^{|x|/(2k)}) \cdot ((x \wedge (\underline{0}^k\underline{1}^k)^{|x|/(2k)}) \ll k)$$

$$\texttt{Collect}(x, k) \equiv x\langle 0\rangle_{|x|/2} \vee (x\langle 1\rangle_{|x|/2} \gg k).$$

Note in the definition of `Spread` that '$\cdot$' denotes concatenation.

We are now ready to explain Algorithm D. Step D.2.2 has the effect of $\oplus$ing the two halves of each key, leaving the second half blank. The subsequent `Collect` packs the $\oplus$ed halves by filling in blank halves. Consequently, each iteration of the main loop halves the length of $|\vec{y}|$. The point of the loop of step D.3 is just to put the $\oplus$ed keys back to the places they originated from. The cost of each iteration of either loop is $O(1 + |\vec{y}|/w)$, so our total cost is $O(\log(b/k) + |\vec{y}|/w)$, as claimed.

LEMMA 4. *We can compute all signatures in linear time.*

Proof. We first pack the n keys in a list $\vec{x} = x_0 \cdots x_{n-1}$, set $\vec{y} = x \oplus \alpha^n$, and apply Algorithm D to $\vec{y}$, producing the list

$$(\texttt{Sig}_\alpha(x_0)\underline{0}^{b-k}) \cdots (\texttt{Sig}_\alpha(x_{n-1})\underline{0}^{b-k}).$$

Since $b = w$ and $k = w/(\log n \log\log n)$, the total running time is linear. ∎

Now, we sort the list $L = \langle(\texttt{Sig}_\alpha(x_i), i) | x_i \in X\rangle$. Each element $(\texttt{Sig}(x_i), i)$ is of length $O(w/(\log n \log\log n) + \log n) = O(w/(\log n \log\log n))$, so by Lemma 2, the sorting is done in linear time. Thereby we group words in X with common signatures. Set $r = 3q(\log n) = O((\log n)^2(\log\log n))$. By Lemma 3 with probability at least $1 - 1/n$, there is no pair (x, y) in any group with $\|x \oplus y\|_1 > r$. For each such group Y, we pick an arbitrary element $y_0 \in Y$. Then no word in $Y \oplus y_0$ has more than $r = O((\log n)^2 \log\log n)$ $\underline{1}$s.

Cluster Busting

For cluster busting, we are given a set Z ($=Y \oplus y_0$ from above) of keys with few $\underline{1}$s. Our general technique is to construct a fast random function f from keys with few $\underline{1}$s to short keys such that

- for any given key with few $\underline{1}$s, f errors with probability $< 1/2$.
- f is 1-1 for all keys for which it is successful.

Thus f will successfully reduce the length of at least half the elements in Z. For the remaining half, we recursively apply a new random function. Due to the halving, the overall complexity is asymptotically the same as that of applying one random function to all the elements.

More specifically, we consider each key divided into $\log n \log\log n$ equal sized *segments*, each of which is divided into r^2 equal sized *pieces*, making up for a total of $q = r^2(\log n)(\log\log n)$ pieces of length $k = w/q$.

Our compression will aim at "hashing" the nonempty pieces (i.e., pieces containing $\underline{1}$s) of a key into the beginning of the key. To make such a function injective, before the hashing, we give each nonempty piece a label identifying its position in the word. This is done as follows in constant time.

$$x := \texttt{Spread}(x,k) \vee (\texttt{Mask}(\texttt{Spread}(x,k),2k) \wedge \prod_{i=0}^{|x|/k-1} (\underline{0}^k \texttt{string}(i,k)).$$

Above, $\prod_{i=0}^{|x|/k-1}(\underline{0}^k\texttt{string}(i,k))$ is only computed once in time $O(|x|/k) = O(\log n)$, which is less than constant time per key. Note that $\log i = \log\log n$ while $k = (\log n)^{\omega(1)}$ by (1), so we have no problems representing i by $k-1$ bits. Also note that $\texttt{Spread}(x,k)$ actually permutes the pieces, but since permutations are injective, this does not conflict with our purpose. The labeling doubles the length of the keys and of the pieces (set $b := 2b$ and $k := 2k$), but this does not affect our asymptotic running times. Now a key is uniquely identified by its set of nonempty pieces.

We will now be more specific about the hashing of the nonempty pieces. For each segment independently, make a random cyclic shift with a multiple of k positions to the right. We consider it an error if any two nonempty pieces end up at the same offset within their segment. Note that there are at most r nonempty pieces, and that there are r^2 different offsets in each segment. Thus, the probability that any two nonempty pieces end at the same offset is at most $1/r^2$, so the probability that any two nonempty pieces collide is at most $\binom{r}{2}/r^2 < 1/2$. If we do not have an error, we complete the compression by taking the bit-wise disjunction of all the $\log n \log\log n$ segments in time $O(\log\log n)$. The resulting key has the same length as

that of a segment, which is $w/(\log n \log\log n)$. Lemma 2 now allows us to complete identifying duplicates by sorting in linear time.

We now need to specify how we will represent the random shifts and how to compute them efficiently. For the representation, we just use one random string β of the same length as the keys. We will use the last $\log r^2$ positions in the ith segment of β to determine how much to shift the ith segment of x. This is done using the function `VectorShift` below. Considering the binary representation of β, the implementation determines which segments needs to be shifted by which powers of 2.

ALGORITHM E. `VectorShift`(β, x, s, k). Takes each segment $x\langle i\rangle_s$ and shifts it cyclically to the right by k times the number of positions represented by the first $\log(s/k)$ bits of $\beta\langle i\rangle_s$.

E.1. For $j := \log k$ to $\log s - 1$ do to:

E.1.1. $\gamma := \texttt{Mask}(\beta \wedge (\underline{0}^{s-1}\,\underline{1})^{|x|/s}, s)$

E.1.2. $x_1 := x \wedge \gamma$

E.1.3. $x_2 := ((x_1 \wedge (\underline{1}^{s-2^j}\,\underline{0}^{2^j})^{|x|/s}) \gg 2^j) \vee ((x_1 \wedge (\underline{0}^{s-2^j}\,\underline{1}^{2^j})^{|x|/s}) \ll (s - 2^j))$

E.1.4. $x := (x \wedge \neg\gamma) \vee x_2$

E.1.5. $\beta := \beta \gg 1$

Given a list $\vec{x} = x_1 \cdots x_n$ of b-bit keys, $\vec{y} = \texttt{VectorShift}(\beta^n, \vec{x}, s, k)$ is computed in time $O(n(b/w)\log(s/k))$. Finally we need

ALGORITHM F. Combines the segments in each key in $\vec{y}$, thus reducing the length to s. If a key has two nonempty pieces at the same offset, error is notified by turning the key into all $\underline{1}$s.

F.1. $b' := b$

F.2. While $b > s$ times:

F.2.1. $b := b/2$

F.2.2. $\vec{y}_0 := (\vec{y} \wedge (\underline{1}^b\underline{0}^b)^{\vec{y}/(2b)})\ \vec{y}_1 := (\vec{y} \wedge (\underline{0}^b\underline{1}^b)^{\vec{y}/(2b)}) \ll b$

F.2.3. $\vec{y} := \vec{y}_0 \vee \vec{y}_1 \vee \texttt{Mask}(y_0 \wedge \texttt{Mask}(y_1, s), b)$ —mark collision

F.2.4. `Collect`$(\vec{y}, b)$

F.3. While $b < b'$: `Spread`$(\vec{y}, b)$; $b := 2b$.

As Algorithm D, this algorithm runs in linear time due to the halving. Thus, we conclude:

LEMMA 5. *Let $\vec{x} = x_1 \cdots x_n$ be a list of b-bit keys, each with at most $r = O((\log n)^{O(1)})$ $\underline{1}$s. In expected time $O(n + n(b/w)\log\log n)$, we can injectively reduce half the keys to length $w/(\log n \log\log n)$.* ∎

Together with Lemma 2, this completes the proof of Theorem 2 under our assumption (1).

5. SHORT WORDS

In this section, we assume that (1) is false, i.e., that $w = (\log n)^{\Theta(1)}$. We let b denote the current key length. Note that if $b = O(\log n)$, sorting can be done in constant time per key using radix sort with character length $\log n$:

LEMMA 6. *We can sort n $O(\log n)$ bit keys in linear time.* ∎

Thus we may assume $b = \omega(\log n)$. Our approach in this section goes as follows. First, we make some improved signatures, allowing us to collect keys in Hamming balls with radius $r = O(\log n/\log b) = O(\log n/\log\log n)$, as opposed to the radius of $O(\log n)$ in the previous section. We then show how to compress keys with r $\underline{1}$s to a length of $O(r \log r) = O(\log n)$ as opposed to the $O(r^2)$ in the previous section. Then radix sort completes the task. Our approach heavily uses that $\log b = O(\log\log n)$ because (1) is false.

Let k be an odd number deviating from $b \log b/\log n$ by at most 1. Let $l = 3 \log n$. For $i = 0, \ldots, l-1$, let $H(i)$ be a set of k independent samples from $\{0, \ldots, b-1\}$. Define the signature $\mathtt{Sig}^H(x)$ of length l such that for $j = 0, \ldots, l-1$,

$$\mathtt{Sig}^H(x)[i] = \bigoplus\{x[j] | j \in H(i)\}.$$

In order to prove that this is a good signature, we shall use

LEMMA 7 [3]. *Suppose that $x_1, x_2, \ldots, x_k$ are* 0-1 *variables which take value* 1 *with probability p. Then* $\Pr(\bigoplus_{i=1}^k x_i = 0) = \frac{1}{2}(1 + (1-2p)^k)$, *which is* $\leq \max\{1 - pk/2, 1/2\}$ *for k odd.*

LEMMA 8. *For H randomly chosen, with probability $1 - 1/n$, for any two keys x and y in X, $|X| = n$, if $\|x \oplus y\|_1 \geq b/k = O(\log n/\log b)$, $\mathtt{Sig}^H(x) \neq \mathtt{Sig}^H(y)$.*

Proof. Let x and y be two keys with $\|x \oplus y\|_1 \geq b/k$. Set $z = x \oplus y$. Then $\mathtt{Sig}^H(z) = \mathtt{Sig}^H_\alpha(x) \oplus \mathtt{Sig}^H_\alpha(y)$. We are searching the probability that $\mathtt{Sig}^H(z) = \underline{0}^k$. For random j, the probability p that $z[j] = \underline{1}$ is $\|z\|_1/|z| = r/b$. Moreover, by Lemma 7, the probability that $\mathtt{Sig}^H(z)[i] = \underline{0}$ is $\leq \max\{1 - pk/2, 1/2\}$. But since $r \geq b/k$, $pk/2 = r/b \cdot k/2 \geq 1/2$. Hence $\max\{1 - pk/2, 1/2\} = 1/2$. Consequently, $\Pr(\mathtt{Sig}^H(z) = \underline{0}^k) \leq 2^{-l} = n^{-3}$. The lemma now follows from the fact that there are only n^2 pairs of keys. ∎

In order to implement $\mathtt{Sig}^H$ efficiently, we will use matrix transposition. Let q divide the word length w. By a *q-word matrix list*, we mean a list of q words $x_1, \ldots, x_q$ representing (w/q) $(q \times q)$-matrices $A_1, \ldots, A_{w/q}$ as follows. The matrices have 1-bit entries and $x_i\langle a\rangle_q$ is the ith row of

matrix A_a. Consequently, entry (i, j) of A_a is found as $(x_i\langle a\rangle_q)[j]$. Recall that transposing a matrix means swapping each entry (i, j) with entry (j, i). Hence, by transposing the q-word matrix list, we mean constructing words $z_1, \ldots, z_q$ such that for any a, i, j, $(z_i\langle a\rangle_q)[j] = (x_j\langle a\rangle_q)[i]$.

LEMMA 9. *We can transpose a q-row matrix list in $O(q \log q)$ time.*

Proof. Let $T(q)$ be the time it takes to transpose a q-word matrix list. Then $T(1) = 0$. For $q > 1$, we first transpose the quadrants recursively. To do this, we split our q-word matrix list $x_1, \ldots, x_q$ into two $q/2$-word matrix lists $x_1, \ldots, x_{q/2}$ and $x_{q/2+1}, \ldots, x_q$. The matrices of the first list are the upper quadrants of the original matrices and those of second lists are the lower quadrants. The two lists are transposed recursively in $2T(q/2)$ time. Combining the results, we get a q-word matrix list $y_1, \ldots, y_q$ representing the original matrices with each quadrant transposed.

We now only need to swap quadrant $(1, 1)$ and quadrant $(2, 2)$ in each matrix. This is implemented as follows. For $i = 1, \ldots, q/2$, $z_i = (y_i \wedge (\underline{0}^{q/2} \times \underline{1}^{q/2})^{w/q}) \wedge ((y_{i+q/2} \wedge (\underline{0}^{q/2}\,\underline{1}^{q/2})^{w/q}) \ll q/2)$, and for $i = q/2 + 1, \ldots, q$, $z_i = (y_i \wedge (\underline{1}^{q/2}\,\underline{0}^{q/2})^{w/q}) \wedge ((y_{i-q/2} \wedge (\underline{1}^{q/2}\,\underline{0}^{q/2})^{w/q}) \gg q/2)$. Now $z_1, \ldots, z_q$ contains the desired transposed matrix.

Swapping the quadrants took $O(q)$ time, so $T(q) \le 2T(q/2) + O(q)$, and hence $T(q) = O(q \log q)$. ∎

A similar matrix transposition is used in [17] for unbounded word lengths.

LEMMA 10. *We can calculate $\mathtt{Sig}^H(x)$ for w b-bit keys x in $O(w + b \log b)$ time.*

Proof. We pack the w keys in b words $\vec{x}_1, \ldots, \vec{x}_b$, each consisting of w/b keys. The words form a b-word matrix list where each row of a matrix represents a key. Using Lemma 9, we transpose the matrices in time $O(b \log b)$. Let $\vec{y}_1, \ldots, \vec{y}_b$ be the resulting words. Now $\vec{y}_j$ contains bit j of every single key.

For $i := 1, \ldots, l$, set $\vec{z}_i = \bigoplus_{j \in H(i)} \vec{y}_j$. This takes time $O(\sum_i |H(i)|) = O(b \log b)$. Now, $\vec{z}_i$ contains bit i of each signature $\mathtt{Sig}^H(x)$.

To extract the signatures, for $i := l + 1, \ldots, b$, we set $\vec{z}_i = \underline{0}^w$, and then we transpose $\vec{z}_1, \ldots, \vec{z}_b$ as a b-word matrix list in $O(b \log b)$ time. The resulting matrix list now has $\mathtt{Sig}^H(x)\underline{0}^{b-l}$ in the position where x was found in the first matrix list $\vec{x}_1, \ldots, \vec{x}_b$. ∎

The keys are now clustered. As in Section 4, for each group Y, we pick an arbitrary element $y_0 \in Y$. Then the maximal number of $\underline{1}$s in any key in $Y \oplus y_0$ is $r = O(\log n / \log b)$. Note that this is optimal, in the sense that there exist less than n different b-bit keys with $\log n / \log b$ $\underline{1}$s.

Cluster Busting for Short Words

We are considering n b-bit keys, each containing at most $r = O(\log n/\log b)$ 1s. Consider each key as divided into pieces of length $k = \log b$, where b is the key length. As in Section 4, we give each nonempty piece a label identifying its position in the key. Each label takes $\log b$ bits, so this doubles the piece and key length. Now each key is uniquely identified by its set of $\leq r$ nonempty segments. As in Section 4, we wish to compress the nonempty pieces into the beginning of the keys, but this time we want a much tighter compression.

First choose a random permutation π of the pieces in a key. Divide $\pi(x)$ into $q = r/(3\log r)$ segments. In each segment, we expect $\leq 3\log r$ nonempty pieces. Using Chernoff bounds (see, e.g., [16]), it follows that the probability that a segment has more than $6\log r$ nonempty pieces is $< 1/r$. Actually, our distribution is not quite Binomial distribution. It would be if each nonempty piece was independently assigned a random segment. However, the dependency introduced by all the segments containing the same number of pieces means that the probability of deviation goes down in the upper tail [23], so the use of Chernoff bounds is still valid. We consider it an error if any segment contains more than $6\log r$ nonempty pieces, so it follows that the probability of error for a given $\pi(x)$ is $< q/r = O(1/\log r) < 1/2$ for r large.

In the following we assume that there is no error. We say a segment is *packed* if all the nonempty pieces lie in the beginning, i.e., if the ith nonempty piece is the ith piece. We now pack segments of exponentially increasing sizes as follows.

- First, consider each piece a segment. Then trivially, each segment is packed.

- Repeat the following for $\log(6\log r)$ rounds: take each even numbered segment and unite with its neighboring segment to the right. To pack the united segment, take the packed nonempty pieces of the right half and shift them to the left so that they become juxtaposed with the packed nonempty pieces of the left part. At the end, each segment is packed and of size $6\log r \cdot k$.

- Repeat the following for $\log(b/q) - \log(6\log r)$ times: as above, take each odd numbered segment and pack its nonempty pieces into its neighboring segment to the left. Since the segments have room for $6\log r$ nonempty pieces, there will be no nonempty pieces remaining in any of the odd numbered segments. Hence all the odd numbered segments may be removed. As a result, the length of the key is halved, and this will be crucial for the overall running time.

• At the end each segment contains $6\log r$ pieces, and we have $q = r/(3\log r)$ segments. Thus, we have compressed all the nonempty pieces in a total of $2r$ pieces. The length of a piece is $O(\log b)$, so the length of the compressed key is $O(r\log b) = O(\log n)$, as desired.

It follows as a special case of Lemma 10 that we can compute $\pi(x)$ for all keys x in time $O(n(1 + b/w \cdot \log b))$. In order to find out how much to shift in connection with the packing, we need a function `VectorCount` that is a kind of reverse to `VectorShift`.

ALGORITHM G. `VectorCount`(x, s, k). Outputs β such that for each segment $x\langle i\rangle_s$, `integer`$(\beta\langle i\rangle_s)$ is the largest number d such that $x\langle i\rangle_s\langle d-1\rangle_k$ is nonempty, or 0 if $x\langle i\rangle_s = \underline{0}^s$.

G.1. $\beta := \underline{0}^{|x|}$

G.2. For $j := \log s$ down to $\log k$:

G.2.1. $x_1 := x \wedge (\underline{0}^{2^j-k}\,\underline{1}^{s-2^j+k})^{|x|/s}$

G.2.2. $\gamma :=$ `Mask`(x_1, s)

G.2.3. $\beta := (\beta \ll 1) \vee (\gamma \wedge (\underline{0}^{s-1}\,\underline{1})^{|x|/s})$

G.2.4. $x := (x \wedge \neg\gamma) \vee (x \wedge (\underline{0}^{2^j}\,\underline{1}^{s-2^j})^{|x|/s} \ll 2^j)$

G.3. Return β

We are now ready to describe the compression.

ALGORITHM H. `Compress`$(\vec{x})$ where $\vec{x}$ is a packed list of keys $\pi(x)$, each of length b. If there is an overflow ($\geq 6\log r$ nonempty pieces) in a segment, it is turned into all $\underline{1}$s.

H.1. $b' := b$

H.2. $s := k$

H.3. While $s < b$:

H.3.1. $\vec{x}_0 := \vec{x} \wedge (\underline{1}^s\,\underline{0}^s)^{|\vec{x}|/(2s)}$

H.3.2. $\vec{x}_1 := \vec{x} \wedge (\underline{0}^s\,\underline{1}^s)^{|\vec{x}|/(2s)}$

H.3.3. $\beta :=$ VectorCourier$(\vec{x}_0, 2s, k)$

H.3.4. $\vec{x} := x_0 \vee$ `VectorShift`$(\beta, \vec{x}_1 \ll s, 2s, k)$

H.3.5. If $s < 6k\log r$, $s := 2s$

H.3.6. Otherwise,

H.3.6.1. $\vec{x} := \vec{x} \vee ((\text{`Mask`}(\vec{x} \wedge (\underline{0}^s\,\underline{1}^s)^{|\vec{x}|/(2s)}, s) \ll s)$ —marking overflow

H.3.6.2. $b := b/2$

H.3.6.3. $\vec{x} := (\vec{x} \wedge (\underline{1}^b\underline{0}^b)) \vee ((\vec{x} \wedge (\underline{0}^b\underline{1}^b)) \ll (b-s))$;

H.3.6.4. `Collect`$(\vec{x}, b)$;

H.4. While $b < b'$: `Spread`$(\vec{y}, b)$; $b := 2b$.

H.5. Return $\vec{x}$.

Step H.3.6.3 permutes the nonempty segments within the keys, but clearly this does not affect the injectivity of the compression. For the first rounds with $s < 6k \log r$, the time bound is $O(n(b/w)(\log(6 \log r))^2) = O(n(b/w)(\log \log \log n)^2)$. Due to the halving, the complexity of the remaining rounds is $O(n(b/w) \log \log \log n)$. That is, the compression is done in $O(n(b/w)(\log \log \log n)^2) = O(n(b/w) \log \log n)$ time. This completes the proof of Theorem 1.

6. BATCHED DICTIONARIES

We will now show how the techniques from the previous sections may be used in the context of dictionaries. Generally, we will define the *dynamic dictionary problem* as follows. For some $b \leq w$, our *universe* $U = \{\underline{0}, \underline{1}\}^b$ consists of all b-bit strings. We wish to maintain a dynamic set $X \subseteq U$, $|X| \leq n$ and a 1-1 index function $\mu : X \to \{0, \ldots, n-1\}$ under the following operations:

`member`(x, X) tells if $x \in X$.

`index`(x, X) returns $\mu(x)$ if $x \in X$; undefined otherwise.

`add`(x, X) if $x \notin X$, defines $\mu(x) \notin \mu(X)$, and adds x to X.

`delete`(x, X) deletes x from X, thus making $\mu(x)$ free for reuse.

Above we are assuming that only linear space $O(n)$ is available. As mentioned in the Introduction, from [3] it follows that on the Practical RAM, there are concrete sets for which a single member ship query takes $\Omega(\sqrt{\log n / \log \log n})$ time on the average. However, here we prove Theorem 3, that we can process a batch of $\log n$ operations on a dynamic dictionary in $O((\log n)(\log \log n)^{1+\varepsilon})$ expected time. Thus, batching of just a few operations gives us an exponential improvement in the per operation cost on the Practical RAM. In the next section we will apply the technique in a basic priority supporting each operation in $O((\log \log n)^{1+\varepsilon})$ expected time, thus showing that priority queues are much easier than dictionaries on the Practical RAM.

In this section, for simplicity, we are assuming a fixed limit n on the number of keys. However, as described in, e.g., [22, Sect. 2.1], we can then also deal with varying n, paying only a small constant factor in the time and space bounds.

Preliminaries

Our approach is based on vector hashing within a word using vector multiplications. In [6] it is argued that Schönhage–Strassen multiplication [19] can be simulated on the Practical RAM to run in time $O((\log w)(\log\log w)^{1+\varepsilon})$. Before going further, we note that we could have used this result for sorting short words with $w = (\log n)^{O(1)}$ as in Section 5. More precisely, we could use this for the multiplication based hashing in the $O(n \log\log n)$ sorting algorithm of Andersson *et al.* [2] and get a Practical RAM bound of $O(n(\log\log n)^{2+o(1)})$. Our $O(n\log\log n)$ sorting is, however, purely combinatorial, and it is faster, matching the best bound using multiplication.

In fact, what we get from [6] is that if words are viewed as vectors of k-bit integers, then coordinate-wise multiplication can be done in time $O((\log k)(\log\log k)^{1+\varepsilon})$. To be more formal, for integers x, y, $|x| = |y|$, and k divides $|x|$, define $\texttt{VectorMult}(x, y, k) = (x\langle 0\rangle_k \times y\langle 0\rangle_k)\cdots(x\langle |x|/k-1\rangle_k \times y\langle |x|/k-1\rangle_k)$ (cf. the definition of $+$ and $\times$ in Section 2).

LEMMA 11 [6]. *Given two words x and y, and k divides w,* $\texttt{VectorMult}(x, y, k)$ *can be computed in time $M(k) = O((\log k)^{1+\varepsilon})$ on the Practical RAM.*

Corresponding to `VectorMult` we will need $\texttt{VectorAdd}(x, y, k) = (x\langle 0\rangle_k + y\langle 0\rangle_k))\cdots(x\langle |x|/k-1\rangle_k + y\langle |x|/k-1\rangle_k)$. Note that if x and y are words, $\texttt{VectorAdd}(x, y, k)$ is easily computed in constant time, as $\texttt{Collect}(\texttt{Spread}(x, k) + \texttt{Spread}(y, k), k)$.

We adopt the following definitions of hash functions. Let $\mathcal{H}$ be a family of functions from $U = \{\underline{0}, \underline{1}\}^w$ to $\{0, \ldots, n-1\}$. We say that $\mathcal{H}$ is *universal* if for all $x, y \in U$, for h chosen uniformly at random from $\mathcal{H}$, $\Pr(h(x) = h(y)) = O(1/n)$. Moreover, $\mathcal{H}$ is said to be *strongly universal* if for all $x, y \in U$, $i, j \in \{0, \ldots, n-1\}$, for h chosen uniformly at random from $\mathcal{H}$, $\Pr(h(x) = i \wedge h(y) = j) = O(1/n^2)$.

LEMMA 12 [7]. *Given a universal family $\mathcal{H}$ of hash functions from U to $\{0, \ldots, n-1\}$, we can maintain a dictionary $X \subseteq U$ with up to n elements such that the expected cost of an operation is the cost of computing a function from $\mathcal{H}$.*

Proof (sketch). A random $h \in \mathcal{H}$ is chosen once for the dynamic dictionary. For each i, we store the set $D[i] = \{(x, \mu(x)) | x \in X, h(x) = i\}$. For any operation involving $x \in U$, we just need to consider $D[h(x)]$, which has expected constant size. Hence the expected cost of an operation is the cost of computing h. ∎

An advantage of strong universal hashing over just universal hashing is that strong universal hashing generalizes nicely to vectors:

Observation 1[7]. Let $\mathcal{H}$ be a strongly universal family of hash functions from $\{\underline{0}, \underline{1}\}^k$ to $\{\underline{0}, \underline{1}\}^k$. For any $x \in \{\underline{0}, \underline{1}\}^w$ and $\vec{h} = (h_0, \ldots, h_{q-1}) \in \mathcal{H}^q$, define $\vec{h}(x) = \bigoplus_{i=0}^{q-1} h_i(x\langle 0\rangle_k)$. Then $\mathcal{H}^k$ is a strongly universal family of hash functions from $\{\underline{0}, \underline{1}\}^{qk}$ to $\{\underline{0}, \underline{1}\}^k$.

Presented in [8] is the following suitable family of strongly universal hash functions:

LEMMA 13 [8]. *Let $k \geq l$ and $a, b \in \{\underline{0}, \underline{1}\}^{k+l}$. Define $d_{a,b} : \{\underline{0}, \underline{1}\}^k \to \{\underline{0}, \underline{1}\}^l$ by*

$$d_{a,b}(x) = (a \times x + b)[0..k-1];$$

that is, viewed as a function on integers, $d_{a,b}(x) = \lfloor((a \times x + b) \bmod 2^{k+l})/2^l\rfloor$. Then $\mathcal{D} = \{d_{a,b} | a \in \{\underline{0}, \underline{1}\}^{k+l}, b \in \{\underline{0}, \underline{1}\}^l\}$ is a strongly universal class of hash functions.

Batched Dictionaries for Short Words

We will now implement batched dictionaries assuming the typical case where $n = w^{\Omega(1)}$. Our basic problem is that we cannot apply Lemma 13 directly to a word x since the multiplication would take too long if, e.g., $w = n$. Instead we view x as a vector of k-bit pieces, for some sufficiently small k, and then hash each piece in parallel using the vector multiplication from Lemma 6. For each piece, we will use the strongly universal hashing from Lemma 13 with $l = k$. Random double words α and β provide vectors of independent $2k$-bit multipliers and adders. Finally, using Observation 1, we get a strongly universal hash function of x if we $\oplus$ the hash values of each piece. A direct implementation of this scheme is presented in Algorithm I.

ALGORITHM I. For α and β, random double words, and x a single word, the algorithm computes a strongly universal hash function $\vec{d}_{\alpha,\beta}$ from $\mathcal{D}^{w/k}$ on x.

I.1. $x_1 := \texttt{Spread}(x, k)$

I.2. $x_2 := \texttt{VectorMult}(\alpha, x_1, 2k)$

I.3. $x_3 := \texttt{VectorAdd}(\beta, x_2, 2k)$.

I.4. $x_4 := \texttt{Collect}(x_3 \wedge \underline{1}^k\underline{0}^k, k)$.

I.5. Return $\bigoplus_{i=0}^{w/k-1} x_4\langle i\rangle_k$.

To see that the above algorithm computes a strongly universal hash function from $\mathscr{D}^{w/k}$, define the permutation $\pi : [w/k] \to [w/k]$ such that $\pi(i) = i/2$ if i even and $\pi(i) = \lfloor i/2 \rfloor + w/(2k)$ if i odd. Then $x_4\langle i\rangle_k = d_{\alpha\langle\pi(i)\rangle_k,\beta\langle\pi(i)\rangle_k}(x\langle i\rangle_k)$, where the functional d is as defined as in Lemma 13. Applying the above algorithm to an individual key would be too slow, so instead we work with a batch of keys in parallel. In particular, we employ Algorithm D for the $\oplus$ing in step I.5:

ALGORITHM J. Given a list $X = x_0 \cdots x_{m-1}$ of keys, each of length $b = O(w)$, and given α, β, $|\alpha| = |\beta| = 2b$, returns $\vec{d}_{\alpha,\beta}(x_0)\underline{0}^{b-k} \cdots \vec{d}_{\alpha,\beta}(x_{m-1}) \times \underline{0}^{b-k}$, where $\vec{d}$ is defined as in Algorithm I.

J.1. $\vec{x}_1 := \texttt{Spread}(x_0, k) \cdots \texttt{Spread}(x_{m-1}, k)$

J.2. $\vec{x}_2 := \texttt{VectorMult}(\alpha^m, X_1, 2b)$

J.3. $\vec{x}_3 := \texttt{VectorAdd}(\beta^m, X_2, 2b)$

J.4. $\vec{y} := \texttt{Collect}(\vec{x}_3\langle 0\rangle_{2b} \wedge \underline{1}^b\underline{0}^b, b) \cdots \texttt{Collect}(\vec{x}_3\langle m-1\rangle_{2b} \wedge \underline{1}^b \underline{0}^b, b)$

J.5. $\texttt{ParallelXOR}(\vec{y}, b, k)$ (Algorithm D)

J.6. Return $\vec{y}$.

Above, steps J.1, J.3, and J.4 take $O(m)$ time, while step J.2 takes $O(\lceil ms/w \rceil M(k))$ time. Finally, Algorithm D in step J.5 takes $O(mb/w + \log(b/k))$ time. Thus we conclude

PROPOSITION 1. *Given a list $x_1, \ldots, x_m$ of m b-bit integers, $s \le w$, and two $2b$-bit integers α and β, in time $O(m + \lceil mb/w \rceil M(k) + \log(b/k))$, we can compute the k-bit strongly universal hash values $d_{\alpha,\beta}(x_1), \ldots, d_{\alpha,\beta}(x_m)$.*

For $n = w^{\Omega(1)}$, $\log b/k \le \log w = O(\log n)$, so combining with Lemma 12, we get

COROLLARY 3. *For $n = w^{\Omega(1)}$, $w/\log n \le b \le w$, we support a batch of $m = \log n$ operations on a dynamic dictionary $X \subseteq \{\underline{0}, \underline{1}\}^b$, $|X| \le n$, in linear space and expected time $O(\log n\lceil (b/w)M(\log n)\rceil) = O(\log n \lceil (b/w)(\log\log n)^{1+\varepsilon}\rceil)$.*

Batched Dictionaries for Long Words

We will now consider the case where $n = w^{o(1)}$. Relating to Proposition 1, this implies that the cost of $O(\log b/k) = O(\log w)$ may exceed $O(\log n)$. In this case, to reduce the cost, we simply change the condition in the while loop D.2 inside Algorithm D called from step J.5 from '$b > k$' to '$b > w/n$'. The loop is then iterated at most $\log n$ times instead of $\log(s/k)$ times, and hence the cost of $O(\log b/k)$ is replaced by $O(\log n)$. The length of the resulting "hash" values then becomes w/n instead of k. Since we

are just stopping the $\oplus$ing earlier, this can only reduce the probability that two different keys get the same hash value, so the probability of such a collision is still $O(1/2^k)$. To deal with the keys of length w/n we employ a completely different technique, resulting in a constant cost per (w/n)-key.

LEMMA 14. *For $n \log n = O(w)$, we support a batch of $\log n$ operations on a dynamic dictionary $X \subseteq \{\underline{0}, \underline{1}\}^s$, $s = w/n$, $|X| \leq n$, in (deterministic) time $O(\log n)$.*

Proof. Let $x_1, \ldots, x_{n'}$, $n' \leq n$, be the sorted list of all current s-bit keys without duplicates. We will, in at most two words, maintain the key-index list

$$\vec{x} = (x_1 \mathtt{string}(\mu(x_1), s)) \cdots (x_n, \mathtt{string}(\mu(x_{n'}), s)).$$

Consider a list $y_0, \ldots, y_m$ of $m = \log n$ s-bit keys. So far we just focus on the index-operation. As pointed out in [2], the technique from [1] allows us to sort m integers of length $\leq w/\log m \log\log m$ in linear time, so in particular, the keys y_i can be sorted in time $O(m)$. Accepting the $O(m)$ cost, below we assume that $y_0, \ldots, y_m$ are in sorted order and that all duplicates have been removed. The idea is now that we first construct the key-0-index list

$$\vec{y} = (y_0 \underline{0}^s) \cdots (y_m \underline{0}^s).$$

For the query signatures that we have already seen, we want to find the index. We do this by merging $\vec{y}$ with $\vec{x}$ in time $O(\log n)$ using the technique from [1]. Now if a query key has been seen before, it is equal to the successor in the list, from which it should take its index. Thus we have the algorithm:

ALGORITHM K. Identifying indices.

K.1. Let $\vec{z}$ be the result of merging $\vec{x}$ and $\vec{y}$, remembering the steps.

K.2. $\vec{m}_1 := (\neg \mathtt{Mask}(\vec{z}, s)) \ll s \wedge (\underline{1}^s \underline{0}^s)^{|\vec{m}_1|/(2s)}$
—make mask for y in list.

K.3. $\vec{y}_1 := \vec{z} \wedge \vec{m}_1$
—extract the $\vec{y}$ in the merged list.

K.4. $\vec{m}_2 := \neg \mathtt{Mask}((\vec{z} \oplus (\vec{y}_1 \gg 2s)) \wedge (\underline{1}^s \underline{0}^s)^{|\vec{m}_1|/(2s)}$
—compare with the successor.

K.5. $\vec{i} := \vec{z} \wedge (\vec{m}_2 \gg s)$ —extract indices.

K.6. $\vec{z}_1 := \vec{z} \vee \vec{i} \ll 2s$.

K.7. Reverse the merge steps, resulting in a list $\vec{y}'$ with updated indices.

When we now traverse $\vec{y}'$ all members of $\vec{x}$ have received their correct index while all nonmembers have index 0. This completes the description of the index operation. If a nonmember is now to be inserted, we just assign it a free index, and all the new keys are merged back in time $O(\log n)$. Deletions of members are done by merging them in with empty indices. Finding out where they end up, we can remove them and their successors in a reverse merge process within the same time bounds.

As a result, we conclude

THEOREM 5. *For $w/\log n \le s \le w$, we can support a batch of $\log n$ operations on a dynamic dictionary $X \subseteq \{\underline{0}, \underline{1}\}^s$, $|X| \le n$, in linear space and expected time $O(\log n\lceil(s/w)M(\log n)\rceil) = O(\log n\lceil(s/w)(\log\log n)^{1+\varepsilon}\rceil)$.*

Theorem 3 then follows with $s = w$.

7. FAST EXPECTED TIME PRIORITY QUEUES

Presented in [22] is an $O(\log\log n)$ time priority queue that, like the $O(n\log\log n)$ sorting from [2], uses multiplicative hashing to achieve linear space. Using batched dictionaries, we will modify the construction from [22] to construct a priority queue for the Practical RAM using linear space and answering queries in $O((\log\log n)^{1+\varepsilon})$ expected time per operation.

Let $T(n, b)$ be the expected time for `insert` and `delete-min` in a priority queue with up to n b-bit integers on a practical RAM. We assume that b but not n is known in advance. We will prove Theorem 4 by showing that

$$T(n, w) = O((\log\log n) + M(\log n)) = O((\log\log n)^{1+\varepsilon}). \tag{2}$$

Define $\ell(x) = \log x \log\log x$. From [2], we have

$$T(n, w/\ell(n)) = O(1). \tag{3}$$

For $b > w/\ell(n)$, we will show that

$$T(n, b) = T(\log n, b) + O(1 + b/w \cdot M(\log n)) + T(n, b/2). \tag{4}$$

As usual, b is assumed to be a power of 2. We can then prove (2) inductively; for applying (4) $\log\log n$ times, and then (3), it follows that

$$\begin{aligned} T(n, w) &= O\left(\sum_{i=0}^{\log\ell(n)} (T(\log n, w/2^i) + 1 + M(\log n)/2^i)\right) \\ &= O((\log\ell(\log n))T(\log n, w) + \log\log T(\log n, w/\ell(\log n)) \\ &\quad + \log\log n + M(\log n)) \end{aligned}$$

$$= O((\log\log\log n)T(\log n, w) + \log\log n + M(\log n))$$
$$= O((\log\log\log n)(\log\log\log n)^{1+\varepsilon} + \log\log n + M(\log n))$$
$$= O((\log\log n) + M(\log n)).$$

In the second to last step we used induction. Thus (2) follows from (3) and (4).

For any b-bit integer x, $\texttt{high}(x) = x[0..b/2-1]$ denotes the most significant half of the bits, and $\texttt{low}(x) = x[b/2..b-1]$ denotes the least significant half of the bits. Thus $x = \texttt{high}(x) \cdot \texttt{low}(x)$ where $\cdot$ still denotes concatenation.

In order to prove 4, we represent a b-bit integer priority queue Q as follows. We require that Q contains at least one key, and we use *min* to store the minimum key in Q. Then $\texttt{find-min}(Q)$ simply returns *min*. Let Q^- be the remaining keys; that is, $Q = \{min\} \cup Q^-$. If Q^- is nonempty, we store it as follows. We have a priority queue H over the high parts of all keys in Q^-; that is, viewed as a set,

$$H = \{\texttt{high}(x) | x \in Q^-\}.$$

Moreover, for each $h \in H$, we have a set $L(h)$ with the corresponding low parts; that is,

$$L(h) = \{\texttt{low}(x) | x \in Q^-, \texttt{high}(x) = h\}.$$

Thus,

$$x \in Q^- \iff \texttt{high}(x) \in H \wedge \texttt{low}(x) \in L(\texttt{high}(x)).$$

If $|Q| = 1$, $H = \varnothing$ and then H is just represented as a nil-pointer. The priority queue Q is initialized with a single key x by $\texttt{alloc}(x, Q)$, which has the affect of allocating Q with $min = x$ and H a nil-pointer. The importance of H just being a nil-pointer is that initiating Q with a single key does not lead to a cascade of recursive calls, but just to a few instructions that can be performed in constant time. If $|Q| = 1$, we can de-allocate Q with $\texttt{de-alloc}(Q)$, also in constant time.

Given the above definitions, it is now straightforward to insert a key x in Q or delete the minimum key from Q.

ALGORITHM L. $\texttt{insert}(x, Q)$.

L.1. If $x < min$, $(min, x) := (x, min)$

L.2. $(h, l) := (\texttt{high}(x), \texttt{low}(x))$

L.3. If $h \in H$, $\texttt{insert}\ (l, L(h))$

L.4. Else,

L.4.1. If $H = \emptyset$, `alloc`(h, H)

L.4.2. Else, `insert`(h, H)

L.4.3. `alloc`$(l, L(h))$

ALGORITHM M. `delete-min`(Q).

M.1. $h :=$ `find-min`(H)

M.2. $min := h \cdot$ `find-min`$(L(h))$

M.3. If $|L(h)| > 1$, `delete-min`$(L(h))$

M.4. Else,

M.4.1. `de-alloc`$(L(h))$

M.4.2. If $|H| > 1$, `delete-min`(H)

M.4.3. Else, `de-alloc`(H)

In order to complete the implementations, we have to specify how to implement $x \in H$. Also, for given h, how do we address $L(h)$? Using standard hashing, these problems are easily solved [13]. Each priority queue Q in the recursive structure gets a unique index i_Q. We maintain a dynamic dictionary over the set X of pairs (x, i_Q) where $x \in Q$. Thus $h \in H \iff (h, i_H) \in X$. We then have *one* array L and interpret $L(h)$ as $L[\mu(h, i_H)]$.

In order to implement `insert`(x) with our batched dictionary from Theorem 5, we introduce as a buffer a priority queue B for up to $O(\log n)$ elements—corresponding to the $T(\log, b)$ term in (4). When an element is inserted, we first insert it into B. When B gets full, for all $y \in B$, we call the batched dictionary with $(\texttt{high}(y), i_H)$, finding out if $\texttt{high}(y) \in H$ and computing the index $\mu(\texttt{high}(y), i_H)$ for the array L. The per element cost is $O(1 + M(\log n)b/w)$. For the sake of `delete-min`, we store the index $\mu(\texttt{high}(y), i_H)$ with y. Thus, at the bottom of the recursion, each key will have a list of $\log\log n$ indices associated with it, one for each recursion level. The advantage is that in `delete-min` the indices are readily available in constant time. Finally, note that with the introduction of B, the implementation of `delete-min` has to be changed so after step M.2, we check which of *min* and $\min B$ is the smaller. If $\min B$ is the smaller, we delete from B; otherwise, we just continue with Algorithm M.

For the buffer B to work in $T(\log n, b)$ time, it is required that we reserve $\Theta(\log n)$ space for B. Since there are $O(\log\log n)$ recursion levels, this leads to $O(n \log n \log\log n)$ space. To reduce the space, inspired by [24], we just divide the keys arbitrarily into sets of size $O(\log n \log\log n)$, each of which we maintain in a standard linear space heap [25] using $O(\log(\log n \log\log n)) = O(\log\log n)$ time per operation. Only the smallest element of each heap is entered in the recursion tree, so the total space becomes linear, and the running time is not affected.

By Theorem 5, the expected per key cost of the batched dictionary is $O(1 + M(\log n)b/w)$. Letting the threshold for when the buffer B is full be a random number $\in \Theta(\log n)$ we make sure that the expected cost is the expected time. Thus we have established (4), and Theorem 4 follows.

8. IMPLEMENTATION WITH THE PENTIUM 4

As mentioned in the Introduction, one of the reasons for considering the Practical RAM is that by avoiding complex instructions like multiplication, it should be easier to increase the word length. In some sense, this is already happening in modern processors like Intel's Pentium 4 [12] (see [20] for a survey of such processors). In addition to working with the usual 32- and 64-bit registers, the Pentium 4 offers a limited instruction set for dealing with 128-bit registers, and this instruction set does not include 128-bit multiplication. However, it does not include 128-bit addition either, so this is not a 128-bit Practical RAM. However, as we shall argue below, the instruction set does support more efficient implementations of the algorithms presented in this paper.

For 128-bit registers, the Pentium 4 only offers vector addition and multiplication on vectors of 8-, 16-, 32-, and 64-bit fields. By restricting ourselves to shorter k-bit fields, we circumvent the complexity of multiplication. For example, the previously mentioned $\Theta(\log w/\log\log w)$ bound on the minimal depth of a polynomial-size circuit for multiplication becomes $\Theta(\log k/\log\log k)$.

Having vector multiplication directly available is a great practical advantage over the Practical RAM implementation from Lemma 11. To quantify this theoretically, we consider *circuit time* where the time it takes to execute an instruction is the minimal depth of a polynomial-size circuit computing it. With this measure, AC^0 operations take constant time, so our AC^0 implementation of vector multiplication from Lemma 11 still takes $O((\log k)^{1+\varepsilon})$ time. However, if vector multiplication is directly available, as on Pentium 4, it only takes $O(\log k/\log\log k)$ time. As a consequence, we only spend $O(\log\log n/\log\log\log n)$ expected time per operation in our batched dictionaries from Section 6. For contrast, we note that the $\Omega(\sqrt{\log n/\log\log n})$ lower-bound from [3] for individual dictionary queries remains valid with circuit time, so batching $\Omega(\log n)$ dictionary operations still gives us a substantial advantage. Finally, with circuit time, we get $O(\log\log n)$ expected time per operation in the basic priory queues from Section 7, matching the $O(\log\log n)$ bound with unit-cost multiplication from [22].

We note that the loss of 128-bit addition does not matter for our algorithms. More precisely, the only place we used addition was in the

implementation of $\texttt{Mask}(x, k)$ (Algorithm A) returning y where $y\langle i\rangle_k$ equals $\underline{1}^k$ if $x\langle i\rangle_k \neq \underline{0}^k$; $\underline{0}^k$ otherwise. We also used addition as part of packed merging from [1] (cf. Lemma 2), but that use was also similar to the mask operation.

However, the Mask operation is nearly directly supported by Pentium 4. More precisely, for $k = 8, 16, 32, 64$, Pentium 4 has an instruction $\texttt{VecNEQ}(x, y, k)$ returning z where $z\langle i\rangle_k$ equals $\underline{1}^k$ if $x\langle i\rangle_k \neq y\langle i\rangle_k$; $\underline{0}^k$ otherwise. Hence $\texttt{Mask}(x, k) = \texttt{VecNEQ}(x, \underline{0}^w, k)$. Similarly, our algorithm can benefit from the local shifts within fields supported by the Pentium 4.

Thus we conclude that a modern processor like the Pentium 4 with its vector-style instruction set is very suitable for implementing our algorithms for 128-bit keys.

Of further research in this direction, the author has recently discovered that some of the more exotic instructions in the Pentium 4 can implement Fredman and Willard's fusion trees and atomic heaps [10, 11]. As interesting consequences, we get the first linear time and space AC^0 implementations of minimum spanning trees and undirected single source shortest paths.

9. OPEN PROBLEMS

Besides the fundamental problem of improving the complexity of sorting, it would be interesting to improve the time bounds in Theorem 2 for identifying duplicates on the Practical RAM. This would not immediately improve the problem of sorting words, but combining with techniques from [4], it would improve Practical RAM string sorting, i.e., the problem of sorting strings, each divided over several words.

Another important challenge is to try implementing some of the algorithms from this paper using a modern processor like the Pentium 4 (cf. Section 8). In [15] it has already been found that similar algorithms were competitive in practice on a normal processor. Therefore it is plausible that we, with the Pentium 4 and careful tuning, could get the world's fastest 128-bit sorting routine.

ACKNOWLEDGMENT

I thank the referee for many good suggestions.

REFERENCES

1. S. Albers and T. Hagerup, Improved parallel integer sorting without concurrent writing, *Inform. and Control* **136** (1997), 25–51.
2. A. Andersson, T. Hagerup, S. Nilsson, and R. Raman, Sorting in linear time? *J. Comput. System Sci.* **57** (1998), 74–93. See also STOC, *ACM* '95.

3. A. Andersson, P. B. Miltersen, S. Riis, and M. Thorup, Static dictionaries on AC^0 RAMs: Query time $\Theta(\sqrt{\log n/\log\log n})$ is necessary and sufficient, *in* "Proc. 37th FOCS, *IEEE*," pp. 441–450, 1996.
4. A. Andersson and S. Nilsson, A new efficient radix sort, *in* "Proc. 35th FOCS, *IEEE*," pp. 714–721, 1994.
5. P. Beame and J. Håstad, Optimal bounds for decision problems on the CRCW PRAM, *J. Assoc. Comput. Mach.* **36** (1989), 643–670.
6. A. Brodnik, P. B. Miltersen, and I. Munro, Trans-dichotomous algorithms without multiplication—some upper and lower bounds, *in* "Proc. 5th WADS," Lecture Notes in Computer Science, Vol. 1272, pp. 426–439, Springer-Verlag, Berlin/New York, 1997.
7. J. L. Carter and M. N. Wegman, Universal classes of hash functions, *J. Comput. System Sci.* **18** (1979), 143–154.
8. M. Dietzfelbinger, Universal hashing and k-wise independent random variables via integer arithmetic without primes, *in* "Proc. 13th STACS," Lecture Notes in Computer Science, Vol. 1046, pp. 569–580, Springer-Verlag, Berlin/New York, 1996.
9. E. W. Dijkstra, A note on two problems in connection with graphs, *Numer. Math.* **1** (1959), 269–271.
10. M. L. Fredman and D. E. Willard, Surpassing the information theoretic bound with fusion trees, *J. Comput. System Sci.* **47** (1993), 424–436. See also STOC, *ACM* '90.
11. M. L. Fredman and D. E. Willard, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, *J. Comput. System Sci.* **48** (1994), 533–551.
12. Intel, The IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference (Order Number 245471), 2001. Available at `http://developer.intel.com/design/pentium4/manuals/245471.htm`.
13. K. Mehlhorn and S. Nähler, Bounded ordered dictionaries in $O(\log\log n)$ time and $O(n)$ space, *Inform. Process. Lett.* **35, 4** (1990), 183–189.
14. P. B. Miltersen, Lower bounds for static dictionaries on RAMs with bit operations but no multiplication, *in* "Proc. 23rd ICALP," Lecture Notes in Computer Science, Vol. 1099, pp. 442–453, Springer-Verlag, Berlin/New York, 1996.
15. S. Nilsson. "Radix Sorting & Searching," Ph.D. Thesis, Lund University, Sweden, 1996.
16. C. H. Papadimitriou, "Computational Complexity," Addison-Wesley, Reading, MA, 1994.
17. V. R. Pratt and L. J. Stockmeyer, A characterization of the power of vector machines, *J. Comput. System Sci.* **12** (1976), 198–221.
18. R. Raman, Priority queues: small, monotone and trans-dichotomous, *in* "Proc. 4th ESA," Lecture Notes in Computer Science, Vol. 1136, pp. 121–137, Springer-Verlag, Berlin/New York, 1996.
19. A. Schönhage and V. Strassen, Schnelle muliplication großer zahlen, *Computing* **7** (1971), 281–292.
20. N. T. Slingerland and A. J. Smith, "Multimedia Instruction Sets for General Purpose Microprocessors: A Survey," Technical Report DCSD-00-1124, University of California, Berkeley, 2000.
21. M. Thorup, Randomized sorting in $O(n\log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations, *in* "Proc. 8th SODA, *AMC-SIAM*," pp. 352–359, 1997.
22. M. Thorup, On RAM priority queues, *SIAM J. Comput.* **30** (2000), 86–109.
23. W. Uhlmann, Vergleich der hypergeometrischen mit der binomial verteilung, *Metrika* **10** (1966), 145–158.
24. P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, *Inform. Process Lett.* **6** (1977), 80–82.
25. J. W. J. Williams, Heapsort, *Comm. Assoc. Comput. Mach.* **7** (1964), 347–348.